Consultative Committee for Space Data Systems

REPORT CONCERNING SPACE DATA SYSTEM STANDARDS

PARAMETER VALUE LANGUAGE — A TUTORIAL

CCSDS 641.0-G-1

GREEN BOOK

May 1992

AUTHORITY

Issue:	Green Book, Issue 1
Date:	May 1992
Location:	CCSDS Panel 2 Meeting, May 1992, Oberpfaffenhofen, Germany

This report reflects the consensus technical understanding of the Panel 2 members representing the following member Agencies of the Consultative Committee for Space Data Systems (CCSDS):

- British National Space Centre (BNSC) / United Kingdom
- Canadian Space Agency (CSA) / Canada
- Centre National D'Etudes Spatiales (CNES) / France
- Deutsche Forschungsanstalt für Luft und Raumfahrt (DLR) / FRG
- European Space Agency (ESA) / Europe
- Instituto de Pesquisas Espaciais (INPE) / Brazil
- National Aeronautics and Space Administration (NASA) / USA
- National Space Development Agency of Japan (NASDA) / Japan

The following observer Agencies also technically concur with this report:

- Department of Communication/Communications Research Centre (DOC/CRC) / Canada
- Institute for Space Astronautics and Science (ISAS) / Japan

This Recommendation is published and maintained by:

CCSDS Secretariat
Communications and Data Systems Division, (Code-OS)
National Aeronautics and Space Administration
Washington, DC 20546, USA

DOCUMENT CONTROL

Document	***Title***	***Date***	***Status/ Remarks***
CCSDS 641.0-G-1	Report Concerning Space Data System Standards: Parameter Value Language -- A Tutorial, Green Book, Issue 1	May 1992	Issue 1 (this document supersedes CCSDS 620.0-G-1)

CONTENTS

<u>Figures</u>

REFERENCES

[1] "Recommendation for Space Data Systems: Parameter Value Language Specification", CCSDS 641.0-B-1, Consultative Committee for Space Data Systems, Blue Book, Issue 1, May 1992.

[2] "Recommendation for Space Data Systems: Standard Formatted Data Units - Structure and Construction Rules", CCSDS 620.0-B-2, Consultative Committee for Space Data Systems, Blue Book, Issue 2, May 1992.

[3] "Report Concerning Space Data Systems: Standard Formatted Data Units - A Tutorial", CCSDS 621.0-G-2, Consultative Committee for Space Data Standards, Green Book, May 1992.

[4] ISO 6093-1985(E) Information Processing - Representation of numerical values in character strings for information interchange.

[5] "Recommendation for Space Data Systems: Time Code Formats", CCSDS 301.0-B-2, Consultative Committee for Space Data Systems, Blue Book, Issue 2, April 1990.

[6] Wells, D. C., Greisen, E. W., and Harten, R. H. 1981, "FITS: A Flexible Image Transport System," Astron. Astrophys. Suppl., 44, 363-370.

[7] Davis, R.L,"Specification for the Object Description Language", Version 1.0, University of Colorado; Boulder, Colorado, 1 June 1990

[8] Johnson, S.C. "Yacc: Yet Another Compiler-Compiler", Computing Science Technical Report No. 32, Bell Laboratories, Murray Hill, N.J., 1975.

[9] Lesk, M. E., and E. Schmidt, "Lex--A Lexical Analyzer Generator", Computing Science Technical Report No. 39, Bell Laboratories, Murray Hill, N.J., 1976.

This page intentionally left blank.

1 INTRODUCTION

1.1 Purpose and Scope

This document describes a keyword-value, the Parameter Value Language (PVL); it provides a description of how and why one would use this language to interchange information. This document provides the rationale for the development of PVL, its intended usage, its format and construction rules as well as suggested practices and cautions.

Most users will be able to use the language after reading this document. Those individuals who are responsible for implementing software associated with the language, or who need more detailed information should also consult Reference [1] which contains the formal specification for the language.

1.2 Context

The need to name values so that they may be referenced later is widely known. Keyword/value notations are one of the most straightforward ways of achieving this goal. For this reason, a number of keyword/value notations have been developed over the years. Because each of these notations was developed to address a specific need, the actual formats of these notations are different. Because these forms seemed intuitive, and are often "parsed" by human users, little thought had been given to developing a formal standard intended for general usage.

Panel 2 of the Consultative Committee for Space Data Systems (CCSDS) is involved with information interchange issues. As part of this work, the CCSDS has developed the Standard Formatted Data Unit (SFDU) concept to improve the automation of information interchange between and among different environments. During the development of this concept, it became clear that the capabilities of a keyword/value notation would be very valuable. It also became clear that since SFDUs would be transferred between heterogenous systems that a simple yet robust notation should be specified.

PVL was developed with these goals in mind. It is a simple, flexible notation which is minimally impacted by typical data transfer mechanisms used between heterogenous systems. To this end, PVL is expressed in American Standard Code for Information Interchange (ASCII) codes.

The development of PVL is built on previous keyword languages such as Flexible Image Transport System (FITS, Reference [6]) and Object Description Language (ODL, Reference [7]). Much of the syntax of PVL is based on ODL, developed for the Planetary Data System at the Jet Propulsion Laboratory. PVL minimizes the semantics of the language to provide implementers the ability to specify the appropriate semantics for their applications. PVL provides syntax and limited semantics. Used in conjunction with a Data Entity Dictionary (DED) it gives users all the capabilities they need in applications that require keyword-value languages.

1.3 How to Use This Document

This document is intended to assist general users in deciding whether PVL is appropriate for their information interchange needs, and if so to provide sufficient information for its effective use.

It is also intended to be an introduction to the formal specification of PVL (Reference[1]). This document consists of the following five section:

Section 1	-	**Introduction** (this section) which provides a brief background for the development of PVL
Section 2	-	**Illustrating PVL Usage** which provides information and examples about how PVL is used, with emphasis on the SFDU environment. A science scenario is used to provide a context for the examples.
Section 3	-	**Forming PVL Statements** which illustrates how PVL statements are formed and gives exampl es of various types of information
Section 4	-	**Suggested Practices and Limitations** which identifies usages which may cause difficulties and suggests conventions which address these concerns.
Section 5	-	**Requirements and Rationale** which contains the list of language requirements on which PVL is based
Annex A	-	**Acronyms and Glossary** which contains a list of the acronyms and glossary of terms used in this document
Annex B	-	**Parser Implementation Advice** which contains a set of suggestions about the implementation of software providing PVL capabilities

2 ILLUSTRATING PVL USAGE

This section gives examples of possible uses of PVL in information interchange. It does so within the context of a scenario demonstrating how a science project could use the capabilities of PVL. Within the data interchange environment, PVL may be used to support a number of common data types. It may also be used in conjunction with specific data dictionaries to form a data description for a data object.

There are several classes of objects that will be used in these examples, each containing a specific type of information:

- Identification objects which contain catalog attribute information about an accompanying object

- Application data objects which contain primary interest data or supplementary data

- Data description objects which contain the descriptive information used to interpret the contents of another object

The scenario presented in the following sections explains how PVL can be used with each of these classes of objects. It is based on an international, multi-year, multi-mission, multi-agency science program. This international program will collect a large amount of data which will need to be managed and distributed over a long period of time.

In order to manage and distribute the large amounts of associated data with this program, the program has chosen to use the capabilities of the CCSDS Standard Formatted Data Unit (References [2] and [3]). Within its usage of SFDUs the program will also utilize many of the capabilities of PVL. These capabilities include, but are not limited to

- The ability to define identifying information about the data collected.

- The ability to provide descriptions of the format of the data.

- The ability to define a data entity dictionary which defines all the parameters used in the project that appear within its data objects.

In order to see how PVL would be used in each of the examples in the scenario, a very brief overview of the language format will be provided. All PVL statements are expressed in ASCII codes. There are two basic constructs that provide the basis of PVL statement formation. They are the assignment statement and the aggregation block. There is also an optional end statement.

An assignment statement has the following general form:

```
Parameter = Value
```

PVL also permits naming collections of statements in the aggregation block construct, which has the following general form:

```
Begin Aggregation  =   Block Name
        ...
        A collection of assignment statements and/or aggregation blocks
        ...
End Aggregation [ =  Block Name]
```

where the brackets indicate that the block name is optional.

In addition, PVL allows the inclusion of comments. Comments begin with a forward slash-asterisk (**/***) sequence, which is followed by the text of the comment. The comment is ended with an asterisk-forward slash (***/**) sequence. A full description of PVL constructs and their formation is contained in Section 3, **Forming PVL Statements**, as well as in Reference [1].

Figure 2-1 illustrates the two major PVL constructs as well as the use of comments. Following line by line in the figure, the first line is a comment that identifies what follows as sample PVL statements. The next line is an assignment statement; it associates the parameter name **Document** with the value **"PVL Tutorial"**. The next series of statements (starting with the parameter **BEGIN_GROUP**) is an example of an aggregation block that associates the block name **UserProfile** with the next four assignment statements. The **END_GROUP** statement closes the aggregation block. In the figure, some of the values are quoted and some are not, although all contain character information. These values are all known as strings and the rules for whether they are quoted are discussed in Section 3.3.2.2. This example illustrates that strings which contain spaces must appear in quotation marks. Also note that each statement ends with a semi-colon. An alternative statement termination is white space. For consistency, the semi-colon is used for all examples in this tutorial. Statement delimitation is discussed in Sections 3 and 4.2.1.

```
/*   Sample PVL statements */

Document = "PVL Tutorial";

BEGIN_GROUP = UserProfile  ;             /* Information used to set up user environment */
        UserType = EndUser;
        Name = "Robert Q. Scientist" ;
        Location = "Moon Base Alpha" ;
        PromptLevel = Novice /* other levels: experienced, expert */;
END_GROUP = UserProfile;
```

Figure 2-1 Examples of PVL Statements

2.1 PVL Usage in Data Interchange

This section illustrates how PVL might be used to interchange several different types of information in a science program such as the scenario described. The examples given are intended to be representative, not to provide full specification of an object. For a given project, each of the three types of objects discussed earlier (identifying objects, application data objects and data description objects) would be fairly common.

2.1.1 Identifying Objects

Data sets may contain objects which provide identifying information about other data. Identifying objects, which might contain information used for cataloging, are often useful in helping users verify that they have the correct set of data. In our science program scenario, decommutated instrument data and summarized key parameter data files are available for distribution from the central data handling facility for a period of time before they are shipped to a permanent archive. There is a catalog which describes which data sets are available. This catalog is implemented by maintaining a database of identifying attributes such as spacecraft name, time span, instrument, file type, file name, etc.

These same catalog attributes are also contained within an identifying object that is included in the distribution of the application data object(i.e., the decommutated instrument data and the key parameters). In our scenario, a PVL-based description for the identifying object is used, since this information needs to be both machine processable and easily readable by humans. A portion of an identifying object, associated with a key parameter file, is illustrated in Figure 2-2.

In this example, the data is from the **PIXIE** experiment from the **POLAR** mission. The starting time is 1 a.m. December 24, 1995. The file type is **KPGS**, and the file name (identified by the parameter **INPUT_FILE**) is **XXXXXND555.W3**. The formation rules for the parameter values are found in an associated project data entity dictionary. Data entity dictionaries contain at a minimum a list of data entity names with their semantic meaning; they may also include information necessary for the validation of values. The time is expressed in the PVL date/time format. The data entity dictionary also contains the formation rules for the parameter **INPUT_FILE**, which includes some encoded information about the file origin that is incorporated into the name. The data entity dictionary would specify that **SPACECRAFT_ID** has a previously designated set of legitimate values of which **POLAR** is one. Similarly, the DED would list **KPGS** among the valid values for **FILE_TYPE**.

```
/* Identifying Information */

SPACECRAFT_ID = POLAR;

INSTRUMENT = PIXIE;

START_TIME = 1995-12-24T01:00:00;

FILE_TYPE = KPGS;

INPUT_FILE = "XXXXXND555.W3";
```

Figure 2-2 Identifying Information using PVL

The use of an explicit DED with PVL is encouraged even when meanings of the parameter names seem obvious, since what is obvious at one period of time may not be as obvious at another. It is also not possible to tell, without the use of a DED, whether a value must be from a designated set of values or otherwise.

The use of PVL to specify a DED is discussed in Section 2.1.3.2.

2.1.2 Use of PVL in Application Data Objects

In our scenario, there are two basic types of application data. One type contains the data of primary interest from the producers' viewpoint. The other type contains supplementary data, which the data producer deems important to also include. This supplementary data may be necessary to interpret the primary data, (such as orbit and attitude) and/or it may be explanatory material.

Within this project, the data of primary interest is of high volume, and thus PVL is a format inappropriate for that portion of the interchange. However, there is often low volume ancillary and supplementary information which needs to be interchanged. PVL could be used to interchange this type of information.

Some of the ancillary information comes in the form of a calibration curve. Figure 2-3 illustrates the use of PVL for describing this calibration curve. The purpose of this curve is to calibrate the temperature sensor raw data to determine the engineering value of temperature. The temperature sensor raw value is collected in three modes (science, dump and engineering), and the location of that data within the telemetry frame changes depending on the mode. The engineering mode data includes multiple readings of the sensor within each major frame.

```
/*  Application Data Example using PVL  */

 SENSOR_NAME = GYRO_A_BASEPLATE_TEMPERATURE;

TELEM_LENGTH = 8;  /* 8 BIT FIELD IN TELEMETRY:  (Raw value of 't')        */

     /*FRAME_POSITION component description:          */
FRAME_POSITION = ((22, 49),       /* Science Mode:    */
                        /*  (CHANNEL, MINOR FRAME)          */
                    (4, 49),                  /* Dump Mode:       */
                        /*  (CHANNEL, MINOR FRAME)          */
                    (45, 6, 1, 6));        /* Engineering Mode:    */
                        /*  (CHANNEL, MINOR FRAME,          */
                        /*   [repeated cycle - in minor frames])*/

POLY_COEFF = (0.106E+3, -0.25E+1, 0.339E-1, -0.264E-3, 0.101E-5, -0.151E-8);
/*   POLY_COEFF description:                    */
/*   POLY_COEFF is of the form: (a0,a1,a2,a3,a4,a5) and  the engineering value   */
/*   of temperature is computed from the  raw sensor value t using:            */
/*                                              */
/*  TEMP <deg C> = a0 + a1*t + a2*t**2 + a3*t**3 + a4*t**4 + a5*t**5 */
```

Figure 2-3 Example of Application Data

This example includes the heavy use of comments to explain how the calibration curve information is intended to be used to calculate the engineering value of temperature from the raw telemetry value. The length of the raw value is given by the parameter **TELEM_LENGTH** and the position of the value in each of three modes is given by the parameter **FRAME_POSITION**. When these two items are applied to the frame data, the raw telemetry value of temperature may be extracted. The equation described by the calibration curve is then applied to convert the raw telemetry value into an engineering value for temperature.

2.1.3 Data Description Information

A Data Entity Dictionary is a repository of the definitions of the vocabulary used in an information system. A DED enables users separated by discipline, geography or time to share the same semantic view of that data. The primary requirement of a DED is to provide complete definitions of the vocabulary used in the data product labels, in the metadata for the data product, and in the descriptive detail of data objects. Each entry in a DED will typically define a data entity, give information such as name, unit, type (e.g., digital, analog, etc.), domain, etc.

In our scenario, the information being interchanged is described through the use of data description objects. Data description objects may contain:

1) information about the physical representation and layout of data,

2) data entity dictionary information which provides definitions of data entities,

3) or a combination of these and other types of information.

Each of these types of data description information may use PVL for its format. Examples of types 1 and 2 are given below. Type 3 includes descriptions where physical and semantic information are intermingled; it may also include information on dependencies or relationships between data objects.

2.1.3.1 Using PVL to Describe a Data Format

PVL can be used to describe the format of general data structures. In the example shown in Figure 2-4, the order of data fields is represented as a sequence, **FLD_ORD**, and each data description is represented by the aggregation block **DATA_REP**.

```
/*  Data Description Example Using PVL  */

FIELD_ORD = (TIME, FLUX, ENERGY_LEVEL);

BEGIN_GROUP = DATA_REP;
        NAME = TIME;
        DATA_SYNTAX = REAL;
        FIELD_WIDTH = 8;
END_GROUP = DATA_REP;

BEGIN_GROUP = DATA_REP;
        NAME = FLUX;
        DATA_SYNTAX = REAL;
        FIELD_WIDTH = 8;
END_GROUP = DATA_REP;

BEGIN_GROUP = DATA_REP;
        NAME = ENERGY_LEVEL;
        DATA_SYNTAX = INTEGER;
        FIELD_WIDTH = 4;
END_GROUP = DATA_REP;
```

Figure 2-4 Data Description Information Using PVL

In this example, only the physical representation of the variables is being described. Other descriptive information would be included in a related data entity dictionary.

2.1.3.2 Using PVL in a Data Entity Dictionary

A Data Entity Dictionary (DED) is a repository of the definitions of the vocabulary used in an information system. A DED enables users separated by discipline, geography or time to share the same semantic view of that data. The descriptive information found in a data entity dictionary associates a name with its definition and other attributes such as a description of its data type, units of measure, or constraints on values.

In our scenario, there are several spacecraft which carry instruments for the science program. Among the items the data entity dictionary provides are a list of the valid values for spacecraft names and their associated instruments. It also contains information on the units of measure and acceptable ranges of instrument readings. Example entries are like those found in Figure 2-5.

```
/*  Data Dictionary Example Using PVL  */

BEGIN_GROUP = ELEMENT_DEFINITION;
    NAME = SPACECRAFT_ID;
    DEFINITION =    "Space craft identifiers for scenario science project.";
    DATA_SYNTAX =    Enumeration;
    DOMAIN_LIST = {WIND, POLAR, GEOTAIL, CLUSTER, SOHO};
END_GROUP = ELEMENT_DEFINITION;

BEGIN_GROUP = ELEMENT_DEFINITION;
    NAME = START_TIME;
    DEFINITION = "Start of coverage time for scenario project file";
    DATA_SYNTAX = Date/Time;
END_GROUP = ELEMENT_DEFINITION;
```

Figure 2-5 Data Dictionary Information Using PVL

In the first entry, the name and definition of **SPACECRAFT_ID** are given. The list of valid spacecraft identifiers is provided in **DOMAIN_LIST**. The **DATA_SYNTAX = Enumeration;** statement indicates that this entry has a specified list of valid values.

In the second entry, the **DATA_SYNTAX = Date/Time;** statement indicates that the value is date/time.

3 FORMING PVL STATEMENTS

This section describes PVL and provides examples that illustrate the formation of PVL statements. PVL assignment statements are generally of the form:

    parameter name = value [;]

where the brackets around the semicolon indicate that it is optional. This format specifies the value to be associated with the parameter name in a simple manner. There are two other statement types: aggregation blocks and the end statement. An aggregation block is a named collection of assignment statements and/or aggregation blocks. A set of PVL statements is known as the PVL Module. The PVL Module exists in an external module such as a file, or SFDU object. The end statement is an optional, special type of statement that indicates the end of the PVL Module prior to the end of the external module. A PVL Module may be empty and contain no statements. Empty PVL Modules may occur within a standard data product which contains several PVL Modules when data for one of the modules was not available.

The following practices may be used to enhance the readability of PVL statements:

- the practice of placing only one statement per line
- the use of block names in end aggregation statements
- the use of an explicit delimiter.

For consistency, the examples in this section will use the semicolon for statement delimitation, will include block names in end aggregation statements, and will present statements on separate lines.

3.1 Character Set

PVL is designed to be used in the transfer of information between heterogenous environments. In order to accomplish this, the character set used must be unambiguously specified. PVL specifies its character set based on the ASCII 7-bit encoding set. It uses the printable ASCII character set, the space character and the following format effector characters: horizontal tab, vertical tab, line feed , carriage control and form feed. The printable ASCII character set includes the lower case letters **a** through **z**, the upper case letters **A** through **Z**, the digits **0** through **9**, and the following non-alphanumeric characters:

! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [\ ] ^ _ ` { | } ~

The space character and format effector characters are referred to as a white space characters. This is because they are often used between statements or statement elements to provide formatting to improve human readability. This formatting of space is said to create "white space" within the display of the statements.

Several characters are reserved for specific purposes within PVL or future use. These characters are referred to as reserved characters which are presented below:

! " # % & ' ( ) , ; < = > @ [ ] ` { | } ~

The reserved characters are not available for use in parameter names, block names, or unquoted strings.

The remaining members of the character set which are neither white space nor reserved characters are referred to as unrestricted characters.

3.2 Comment

PVL allows the use of comments to provide explanatory material within the PVL Module. Comments do not impact the association of names with values, and are ignored by that process. Comments promote human readability and understandability of PVL statements, and can provide additional contextual information for the human reader. Comments are a sequence of PVL characters which begin with a forward slash asterisk character pair (**/***) and end with a asterisk forward slash character pair (***/**). Comments may not be nested. This means that the comment ends with the first asterisk forward slash (***/**) character pair encountered within a comment string.

Examples of valid comments:

```
/*This is comment one *//* This is comment two*/

/*  This is a long comment which has the format effectors
        carriage return, line feed, and tab inside it*/
```

Example of invalid comment:

```
/*this /* is not a comment */
```

The second forward slash asterisk pair is not allowed within the comment statement and would be considered an error.

3.3 Assignment Statement

The assignment statement provides the mechanism to associate a value with a parameter name. The assignment statement consists of a parameter name followed by an assignment symbol (equal sign), and a value. Any value may optionally be followed by an units expression. The statement ends with an explicit delimiter (the semicolon), the beginning of the next statement, or the end of the PVL Module. If a statement is delimited due to the beginning of the next statement, then white space, comment(s) or a combination of white space and comment(s) must occur prior to the next statement. White space and/or comments are ignored if they occur between the parameter name and the assignment symbol; the assignment symbol and the value; the value and the optional units expression; the value and the explicit delimiter; the units statement and the explicit delimiter; or the explicit delimiter and the beginning of the next statement. Explicit delimiters are only allowed at the end of a PVL statement, hence null statements are illegal (e.g., both **VAR1 = A;;** and **VAR2 = ;** are errors).

3.3.1 Parameter Name

A parameter name is the name used to reference the value. The parameter name may be any combination of unrestricted PVL characters. This means that white space characters and reserved characters may not be used in parameter names. Additionally, parameter names may not contain the character combinations of forward slash/asterisk (/*) or asterisk/forward slash (*/), since these are used as comment delimiters. It must also be distinguishable from a numeric or date/time value.

A parameter name should be chosen for human readability. If a data entity dictionary is used in conjunction with PVL, additional descriptive information may be recorded in the data entity dictionary.

The following are examples of valid parameter names:

SPACE_CRAFT   StartTime   PHASE.2.4

The following are NOT valid parameter names:

SPACE CRAFT -- contains white space character
Start/*Time -- contains comment begin characters
PHASE[2,4] -- contains reserved characters
PHASE(2) -- contains reserved characters

3.3.2 Value

The value portion of an assignment statement may contain either a simple value, a set or a sequence, any of which may be followed by a units expression. A simple value is a numeric, a string or a date/time value. Sets and sequences are a collection of zero or more other values, any of which may be a set or sequence itself. A Sequence is a collection of values in which the order of the values has meaning. A Set is a collection of values for which the order is not important. A value must be provided in an assignment statement (e.g, **VAR =;** is an error).

The types of values allowed within the assignment statement are classified as being simple values, sets, or sequences. A simple value may be a numeric, a string, or a date/time value. Simple values can be viewed as a single occurrence of a number or sequence of characters. Sets and sequences are collections of values.

3.3.2.1 Numeric Values

Numeric values are composed of PVL characters which follow the following encoding rules. Numerics are widely used within the space science and engineering communities. Numerics in PVL may be in either decimal or nondecimal notation.

A decimal number in PVL uses one of the three numerical representations (integer, floating point and exponential) specified in ISO 6093 (Reference [4]). These representations use the following subset of the PVL characters set: the digits (**0-9**), the period (**.**), minus (**-**), plus (**+**), **e** and **E**.

Figure 3-1 contains examples of decimal numerics to illustrate the three different forms. Integers consists of a sequence of digits, which may optionally be preceded by a sign. If no sign is present, then the value is assumed to be positive. In the first example, the value of **Records** contains a positive integer. The second example, the value of **OFFSET** contains a negative integer.

Floating point numbers are represented as a sequence of digits with an embedded decimal point. Floating point numbers may be optionally preceded by a sign. Unsigned floating point numbers are assumed to be positive. In the third example, **LONGITUDE** contains a negative floating point number. In the fourth example, **PITCH** contains a positive floating point number with an explicit sign.

```
/*        Integer Numerics                */

Records = 4;

OFFSET = -2000;

/*        Floating Point Numbers          */

LONGITUDE = -59.7;

PITCH = +17.65;

/*        Exponential Numbers             */

Flux = 3.2E-2;

ALTITUDE = 2.56e6;
```

Figure 3-1 Examples of Decimal Numerics

Exponential numbers consist of a significand (i.e., mantissa) and an exponent separated by the letter **E** or **e**. The value of the number equals the value of the significand multiplied by the result of 10 raised to the power represented by the exponent. The significand may be either an integer or a floating point number. The exponent must be an integer. Either the significand or the exponent may be negative. In the fifth example **Flux** contains a positive exponential number with a negative exponent. In the sixth example, **ALTITUDE** contains a positive significand with a positive exponent. Note that the difference in case of the exponent indicator **E** in the last two examples is a matter of style, and does not affect the interpretation of the value.

Some applications require nondecimal numerics. The nondecimal encoding forms allow the use of bases other than ten for numerics. Nondecimal encoding is only allowed for integers. The supported nondecimal forms are binary, octal and hexadecimal. The nondecimal notation used consists of an optional sign, followed by the radix (expressed in base 10), a number sign (#), the nondecimal number and the number sign. The nondecimal form itself is interpreted as being positive and uncomplemented.

Nondecimal numeric encodings provide an alternate method of providing numeric information for conversion into local numeric form. If the application intends to use nondecimal encodings without conversion, such as for flag settings or masks, then the values should generally be presented as quoted strings. Quoted strings will be passed unprocessed to the application for its own uses. An application may use nondecimal numerics to pass flag values if certain precautions are taken. (See Section 4.5.3)

Nondecimal integers may be binary, octal or hexadecimal, using bases of two, eight and sixteen respectively. Figure 3-2 contains examples of nondecimal numerics to illustrate the encoding rules.

```
/*      Binary Integers         */

CODE_REP = 2#0010111011110000#;

OFFSET = -2#0101#;

/*      Octal Integer           */

StatusCode = 8#3372#;

/*      Hexadecimal Integer     */

FluxMagnitude = 16#AF07E619#;
```

Figure 3-2 Examples of Nondecimal Integers

Binary numerics are expressed in base two, therefore the only digits available for use within this representation are **0** and**1**. The radix value for binary numbers is **2**. In the first example, **CODE_REP** shows a positive binary number. In the second example, **OFFSET** shows a negative binary number, which would convert to -5 in decimal notation. Note that the sign of the value precedes the radix value, all nondecimal encodings are positive and uncomplemented to allow for transport between systems with different internal representations of negative integers.

Octal integers may only use the digits **0-7** for their nondecimal representation. The radix for octal numbers is **8**. In the third example, **StatusCode** shows a positive octal number.

Hexadecimal integers may only use the digits **0-9**, and the letters **a-f** or **A-F** for their nondecimal representation. The radix for hexadecimal numbers is **16**. In the fourth example, **FluxMagnitude** is assigned a positive hexadecimal value.

3.3.2.2 String

A string value consists of a sequence of PVL characters. A string value may be either quoted or unquoted. An unquoted string may only contain unrestricted PVL characters. A string which requires reserved characters or white space characters must be quoted. A string must also be quoted if it could be interpreted as a numeric when its intended usage is as a string.

A quoted string consists of a sequence of zero or more PVL characters enclosed in matching quote string delimiters. The quote string delimiters are the quotation mark (**"**) and the apostrophe (**'**).

The only restriction on the contents of a quoted string is that it may not use the quote delimiter, which encloses the string, within the string. Quoted strings are required when a string value contains white space or reserved characters. A string must also be quoted if the value could be interpreted as a numeric or date/time value, and a string value is intended.

Quoted string values may include either an apostrophe or a quotation mark, as long as the alternate quote delimiter is chosen, e.g., a string enclosed within quotation marks may include apostrophes.

```
/*        Examples of Quoted Strings      */

Remark =  "This is a free form string, containing reserved and white space characters!";

ID_CODE = '3.5E1';

Event = "Halley's Comet";

Empty = "";

SPACE_CRAFT = 'WIND';

Quote1 = "John said 'Goodbye' and then left.";

Quote2 = 'John said "Goodbye" and then left.';
```

Figure 3-3 Examples of Quoted Strings in PVL

Figure 3-3 contains several examples of quoted strings. In the first four examples the string is required to be quoted. In the first, reserved and white space characters were included in the string. In the second, the identifier could be interpreted as a numeric. In the third, white space and apostrophe are included in the value, which requires that the string be enclosed in quotation marks. In the fourth example, the user needed to represent an empty string as the value. The fifth example is a quoted string which could alternately be presented as an unquoted string; it is quoted as a matter of preference. The last two examples illustrate the use of quote delimiter characters enclosed within strings delimited by alternate quote delimiter.

```
/*        Examples of Unquoted  Strings */

SPACE_CRAFT = WIND;

EMAIL = AA::BBBBB;
```

Figure 3-4 Examples of Unquoted Strings

A string may take the unquoted form if it meets certain criteria. An unquoted string may be of any length greater than zero, and consists of a sequence of unrestricted PVL characters. They may not contain white space, reserved PVL characters or comment delimiters (**/*** or ***/**). In addition they must not conform to any of the numeric encoding rules since they would be interpreted as numerics rather than strings. Figure 3-4 contains examples of unquoted strings. In the first example, **SPACE_CRAFT** includes only alphabetic characters. In the second example, **EMAIL** contains both alphabetic and nonrestricted symbol characters.

3.3.2.3 Set

A set is collection of values for which order of the elements is not important. Sets may contain simple values, sets, sequences or any combination thereof. Generally, sets contain information of the same type, but this is not a requirement. A set is delimited by curly brackets. A set may be empty or it may contain one or more values. If a set contains more than one value, the values are separated by commas. When commas are used, a value must appear both before and after the comma (i.e, the statement **set1 = {2,,};** is illegal). White space or comments may occur between the curly brackets, values and/or commas. Figure 3-5 shows several examples of assignment statements containing sets. Note that the last two examples are equivalent, since the positional order of a set's elements are not significant.

```
/*        Examples of Sets        */

FLAGS_SET = {/*empty set*/ };

INSTRUMENT_IDS = {PIXIE}; /* set with one element */

FILTERS = {RED,
           BLUE,
           GREEN};

VALID_RANGES_1 = {(0,50), (51,100), (101,200)};

VALID_RANGES_2 = {(51,100), (0,50), (101,200)};
```

Figure 3-5 Examples of Set Values

3.3.2.4 Sequence

A sequence is a collection of values for which the order in which the elements are given by user is considered important. Sequences are delimited by a matching parenthesis pair. Sequences may contain simple values, lists or both. Sequences often contain values of more than one type. If a data entity dictionary is used to provide descriptive information, it may prescribe what type of value is valid at each position in a sequence. A sequence may be empty, or may contain one or more values. If more than one value is enclosed in a pair of parentheses, they are separated by commas. When commas are used, a value must appear both before and after the comma (i.e, the statement **sequence1 = (2,,);** is illegal). White space and/or comments may occur between the parentheses, values, and/or the commas separating values.

```
/*         Examples of sequences            */

START_TIMES = (); /* empty set -- no observations */

Instruments = (PIXIE);  /* Single entry */

EnergyLevels = (0, 10, 1000, 10000, 100000);

ObservationType = (POLAR, PIXIE, 5, 'Definition');

LatLon_1 = ((0,0), (0,10), (0,20));

LatLon_2 = ((0,10), (0,0), (0,20));
```

Figure 3-6 Examples of Sequences

Figure 3-6 contains some examples of sequences. Note that the first example is an empty sequence. The second example contains a sequence with a single entry. The third example shows a case where all the enclosed values are of the same type. The next example shows that elements of a sequence need not be of the same type. The fifth and sixth examples are of sequences of sequences. Note that although the sequences in the last two examples contain the same elements, that they are not equivalent because they are not in the same positional order, and that order is significant for sequences.

3.3.2.5 Date/Time Value

Date and time values are used in number of different applications and are often represented in a myriad of different formats. PVL uses a subset of the CCSDS ASCII time code format. This provides a standard time/date representation to be used across applications. All times are expressed as Universal Coordinated Time, which is also known as Greenwich Mean Time.

Date and time can be presented as independent units or combined into date/time value. The date format may indicate year, month and day, or year and day of year. In each case the year is represented by a four digit sequence. At least one digit of the sequence must be non-zero.
For the year, month and day format the form is

```
   yyyy-mm-dd
        where  yyyy is the four digit year (0001-9999)
                    mm is the two digit month (01-12)
                    dd is the two digit day (01-31)
```

For the year and day of year format the form is

```
   yyyy-ddd
        where  yyyy is the four digit year (0001-9999)
               ddd is the three digit day of year (001-366)
```

Time is represented in the following form

```
    hh:mm[:ss[.d...d]]
        where  hh is the two digit hour (00-23)
               mm is the two digit minute (00-59)
               ss is the two digit second (00-60)
```

```
/*   Examples of Date/Time      */

StartTime = 1994-12-01T13:12;

EndTime = 1994-336T13:12:00.567Z;

Effective_Date = 1994-03-20;

BackUpBegin = 22:30:00.000Z;
```

Figure 3-7 Examples of Date/Time Values

Figure 3-7 provides examples of the various date/time forms. The first example uses the full time format using the year-month-day form of date and a minimum length time code, without the optional terminator symbol. The second example uses the full date/time code with the date in year-day of year form, the full precision form of time, and the optional terminator. The third example shows the date subset form using the year-month-day form. Notice that the **T** separator is not used in this form. The fourth example uses the time subset.

3.3.2.6 Units Expression

It is often helpful to provide information about the units involved with simple values, sets or sequences. A units expression consists of a string enclosed within angle brackets (**<>**). The units value is the sequence of characters between the angle brackets excluding any leading or trailing white space characters.

PVL allows a units expression to optionally follow a set, a sequence or a simple value. A units expression may follow a value enclosed within a set or sequence. Only one units expression may follow any simple value, set or sequence.

PVL passes unit information to an application. The application does all interpretation of the units and sets precedence in cases where a member of a set or sequence has a units expression, and the list has a units expression. The application also determines if a units expression overrides units which may have been set by a data entity dictionary.

Units are provided by PVL as a service to the calling application which may or may not choose to use them, as appropriate.

Figure 3-8 contains examples of units expressions in assignment statements. Note that in the first example, the units follow an individual numeric. In the second case, two separate numerics are included in a sequence each with its own units expression. In the third case, the units expression following the sequence applies to the sequence contents. The fourth case illustrates that reserved characters and white space characters may used as part of the units value. Only the angle brackets themselves are disallowed within the units value. As with quoted strings, the sequences forward slash-asterisk and asterisk-forward slash are treated simply as characters within the units value.

```
/*  Examples using units expressions  */

Velocity = 3000< kps >;

TEMP_LOG = (+357<sec>, 32<K>);

Flux = (357, 300, 550)<T>;

Growth = 75 < % change>;
```

Figure 3-8 Examples of Units Expressions

Note that while units are generally associated with numeric values, they are not restricted to numeric values. The application determines the interpretation of a unit following any set, sequence, or simple value.

3.4 Aggregation Block

It is often useful to note that a group of values is related, and to be able to refer to such a group by name. For example, the aggregation block may be used to describe data entities in a data entity dictionary.

The aggregation block allows the grouping of statements and/or other aggregation blocks into a named block. The block begins with a start aggregation statement. The statement(s) and/or nested block(s) follow, and the block is then closed with an end aggregation statement. Aggregation blocks may be designated as either groups or objects. Both designations are in common usage. PVL makes no distinction between the two types of aggregation blocks. Applications may make use of both types and assign any distinction in meaning or use that they deem appropriate.

A block name is required for the begin aggregation statement and is recommended for end aggregation statements. Begin aggregation statements are of the form:

```
BEGIN_GROUP = block name
  or GROUP  = block name
 or
BEGIN_OBJECT = block name
  or OBJECT  = block name
```

End aggregation statements are of the form:

```
END_GROUP [= block name]
        or
END_OBJECT [= block name]
```

where the use of square brackets shows that block names are optional in the End Aggregation statement.

The type must match in the begin and end aggregation statements. If a block name is used in the End aggregation statement, it must match the name in the paired begin aggregation statement. Aggregation blocks may be nested, but inner blocks must be closed before any enclosing outer blocks are closed.

This means that the aggregations of the form:

```
    begin aggregation = Name 1
   ...
            begin aggregation = Name 2
            ....
    end aggregation = Name 1
            end aggregation =Name 2
```

are ILLEGAL, since the inner aggregation block (Name 2) had not been ended before the enclosing block (Name 1). The indentation is used to show the nesting level.

Figure 3-9 contains examples of valid aggregation blocks. All examples include block names in the end aggregation statement since this is the preferred practice. One of the examples illustrates how an aggregation block can be used as a data entity dictionary entry (Section 2.1.1). The other example shows how an aggregation block can be used to describe an image type. Note that group and object aggregation blocks may be nested within each other.

```
/*      Examples of Aggregation statements     */

BEGIN_GROUP = ELEMENT_DEFINITION;
        NAME = SPACECRAFT_ID;
        DEFINITION =  "Space craft identifiers for scenario science project.";
        DATA_SYNTAX_ID = C;
        DOMAIN_LIST =       {WIND, POLAR, GEOTAIL, CLUSTER, SOHO};
END_GROUP = ELEMENT_DEFINITION;

BEGIN_OBJECT = IMAGE_DEF;
        BEGIN_GROUP = SIZE;
                N_ROW = 512;
                N_COL = 512;
        END_GROUP =SIZE;
        FILTERS = {BLUE, RED, GREEN};
END_OBJECT = IMAGE_DEF;
```

Figure 3-9 Examples of Aggregation Statements

3.5 End Statement

The PVL Module may be ended prior to the end of the provided sequence of octets within which it is presented by the use of the End statement. The End Statement is a special type of statement and has different delimitation rules from other statements within PVL. In particular, the End Statement is delimited by one of the following constructs: the first white space character following **END**, a semicolon immediately following **END**, the closing of a comment immediately following **END** or the end of the provided octet space immediately following **END**. The End Statement may also be used to promote readability in the SFDU environment, when marker delimitation is used, by alerting the human reader that the PVL Module has ended.

Figure 3-10 illustrates the use of the End Statement. The statement **Filter = Blue;** is the last assignment statement in the PVL Module. In the construct **END ;/*That's All!!*/**, the space following **END** terminates the PVL Module, and that the semicolon is the first character of Non-PVL data available to an application for its own uses. Note that even though the data following **END** looks like a series of PVL statements, it is outside of the PVL Module and therefore it is to be processed separately by the application environment.

```
/*        Example using End Statement  */

Filter = Blue;

END ;/*  That's All!!!*/

BEGIN_OBJECT = TABLE

          Items = (123, 444, 555, 7777)

END_OBJECT
```

Figure 3-10 Example of an END Statement

4 RECOMMENDED PRACTICES AND USAGE LIMITATIONS

PVL is intended to enable both human readability and machine parsability. Some of the suggested practices are intended to enhance human readability without sacrificing machine parsability. Other practices are intended to minimize problems associated with various transfer techniques, or to assist applications in their use of PVL values. This section also points out known limitations within the language.

4.1 Data Entity Dictionary Usage

Since PVL provides syntax with a limited set of semantics, the semantics for interpreting PVL statements must be provided externally. The most common way of providing semantics for a PVL Module is by the use of an associated DED. As discussed in Section 2.1.3.2, a DED is a repository containing information necessary for the interpretation of parameters defined in a PVL statement. It contains, at a minimum, a list of data names and their definitions. It may also contain information on the validation of parameter values with information such as range, data type, formation rules, or a list of valid values. It may also contain information on the interrelationship of data entities.

A DED may also contain information on application conventions, such as lexical rules for parameter names or possible semantic meaning associated with use of **GROUP** or **OBJECT** as the keyword used in the construction of aggregations.

The use of a DED in conjunction with PVL is highly recommended to provide a fuller set of semantics for the interpretation of PVL statements.

4.2 Human Readability Practices

The following practices are aimed at improving the human understanding of PVL statements. These practices are not required, but serve to make the information more readily accessible to the human user.

4.2.1 Statement Separation

The ability to separate statements is vital to understanding the information contained in a PVL Module. It is recommended that

1) The same statement delimitation technique is used throughout any given PVL Module.
2) Explicit statement delimitation is the preferred form to ensure reliable interchange between different environments. This is particularly the case where transfers take place between different text environments, in which ASCII control characters may be transferred in different ways and in which spaces characters can be removed by underlying service software.
3) If implicit delimitation is used, statements should appear on separate lines to enhance readability.

The preferences of the intended user community and the environments in which the PVL Module is expected to be found may provide the basis for deciding what particular delimitation technique is appropriate.

4.2.2 Aggregation Block Readability

In the definition of aggregation blocks, the use of indention between the begin and end block statements greatly improves readability. Since blocks may be long and nested, the use of these visual clues allows the human user to more readily identify the structure and nesting of aggregation block statements.

The use of block names in the end aggregation statement also aids in the readability of aggregation blocks. Its use aids the reader in determining the scope of long, complicated aggregations, particularly if there is nesting.

It is recommended that one of each set of begin-aggregation keywords be used in any given module, e.g., don't mix the use of **BEGIN_GROUP** and **GROUP** within a module. Mixing synonym types might cause a user unnecessary confusion.

Figure 4-1 illustrates how the use of indentation and the use of block names in the end aggregation statement makes aggregation blocks more readable. The first example uses the preferred methods, which increases the visibility of the structure of the information. The second example contains the same information with a different block name, but without the use of indentation, block names in the end aggregation statement, or consistent begin aggregation keyword forms. While the second example is legal within PVL, it is much more difficult for a human reader to understand.

Both examples will be properly parsed by a machine. However, if there had been an error in the structure, the first example would be easier to examine and find the error.

```
/* Recommended Aggregation Style  */

BEGIN_GROUP = FirstGroup;
        BEGIN_OBJECT = Object1;
                Line1 = 1;
        END_OBJECT = Object1;
        STUFF = "Some other information";
        BEGIN_OBJECT = Object2;
                Field1 = A;
                Field2 = B;
        END_OBJECT = Object2;
        MORE_STUFF = "Yet more information";
END_GROUP = FirstGroup;

/* Legal, but not recommended  */

BEGIN_GROUP = SecondGroup;
OBJECT = Object1;
Line1 = 1;
END_OBJECT;
STUFF = "Some other information";
BEGIN_OBJECT = Object2;
Field1 = A;
Field2 = B;
END_OBJECT;
MORE_STUFF = "Yet more information";
END_GROUP;
```

Figure 4-1 Aggregation Style Example

4.2.3 Use of Meaningful Parameter Names

The use of meaningful parameter names greatly increases the human readability of PVL statements. While the statement **VAR1 = POLAR;** is easily parsable, it is less meaningful than the statement **SPACE_CRAFT = POLAR;** Since PVL statements are often used with identifying objects in the SFDU environment, it is important to choose parameter names that will facilitate the identification of the parameters even in the absence of the associated data entity dictionary.

4.2.4 Use of Comments

Use of comments within PVL can aid the human reader in understanding the data. However, the thoughtless use of comments may hinder rather than assist human readability by producing a cluttered appearance. Common uses of comments are to

- describe PVL Module structure,
- describe aggregation blocks, or
- explain individual statements.

The use of special syntax within comments which is intended to be machine interpretable is severely discouraged.

4.3 Application Conventions

This section discusses various conventions that application developers may find useful. It focuses on the interface between information in PVL form and the application which will be using it.

4.3.1 Parameter Name Conventions

To facilitate the direct use of PVL parameter names within an application, the parameter names should follow formation rules specific to the application, project or discipline.

For example, within certain environments the hyphen is reserved as an operator. In other environments, the hyphen is used to enhance the readability of the variable names. In cases where the PVL Module may be used in several different application environments, it may be necessary to use a highly restrictive set of formation rules in the construction of parameter names.

4.3.2 Use of Group and Object Aggregation Types

PVL allows for the use of both group and object as designations for aggregation blocks. If an application does not need to distinguish between aggregation types, a single convention should be chosen, i.e., all aggregations should be designated as group or object.

Some applications have found it useful to differentiate between aggregation types. As an example, an application may find it useful to designate the **OBJECT** keyword to define a structures of items, and the **GROUP** keyword to collect together a single instance of related items.

4.3.3 Definition of a Null Value

There are some cases where it is useful to note that there is no valid value for a parameter. While it is possible to denote an empty quoted string, set or sequence by providing no value between delimiters, this is not the case for parameters whose values are expected to be unquoted strings, numerics or date/time values. Some applications may find it useful to define a specific value in their data entity dictionary to denote no valid value exists. The value **NULL** is often used for this purpose. The use of a null keyword is especially useful in applications where a template is used, to verify that the parameter was not simply overlooked.

4.3.4 Use of End Statement

Some application may wish to have additional data included with the PVL Module in the same group of externally provided octets. The End Statement allows a way to indicate the end of the PVL Module prior to the end of the externally provided sequence of octets. Other applications may wish to use the

End statement to stop PVL processing when the provided octet set is substantially larger than the sequence of PVL Statements.

4.4 Transfers of PVL Statements

PVL is designed to be transferred between heterogenous systems. As noted previously, PVL consists of a defined set of octets (e.g., a file, an SFDU), in which PVL statements are found. The hardware and software services that are used in data transfer are called transfer protocols. Transfer protocols exist for physical media (e.g., tapes and CD-ROMs) on which data is placed and for communication networks over which data may be transferred. There are two types of transfer: binary and text.

A binary transfer is defined as an octet by octet copy of the original sequence of octets. The set of octets is strictly preserved, and the data is not altered in any way. The internal representation of the data is the same on the recipient system as it was on the sending system. In general, transfers to media (e.g., CD-ROM and tape) are binary transfers.

In text transfers, the sequence of octets is assumed to be a sequence of characters intended for human interpretation. This assumption means that certain transformations may take place to assure that the conventions for character representation and end of line match what the recipient system expects text to look like. For example, an end of line is represented on some machines as a single character (line-feed), on other machines end of line is represented as a two character sequence (carriage-return and line-feed), and on some mainframes the line indication is an embedded character count at beginning of each line in a file. Text transfers can transform the sequence of octets to make sure that lines are preserved in a manner that the receiving system understands. Text transfers may also make assumptions about the maximum number of octets in a line, and may trim trailing blanks from a line. Text transfers will also convert character representations to the native form of the receiving system (e.g., ASCII to EBCDIC). Text transfers generally occur in protocols which rely on communications networks since the transformations depend on knowing the text conventions in the receiving environment. Transfers to transportable media have no assurance of what the receiving system will be, therefore any transformations occur outside of the transfer protocol itself, i.e., when the media concerned is read.

The choice of transfer protocol may be set by application convention or the nature of context in which the PVL is being transferred. For example, if the PVL Module is contained within an SFDU which also contains binary data, the entire SFDU will need to be transferred using a binary protocol.

Text transfers are, for the above reasons not ideal for the transfer for PVL between heterogenous systems. However, text transfer of PVL can be done successfully. This section illustrates text transfer and explains what actions a user can take to counteract some of the pitfalls.

4.4.1 Line Format Differences

PVL Modules environment may employ line formatting to assist human user readability. Since different systems use different representations for end of line, a binary transfer may result in the PVL Module looking different in the receiving environment. Figure 4-2 illustrates how the differences in the representation of end of line can effect the appearance of the PVL Module. The examples show the effects of a binary transfer between a system using line-feed to denote end of line to a system using the sequence carriage-return/line-feed to denote end of line.

```
/* Set 1 */

Line1 = "This is line 1";
Line2 = "This is line 2";

/*  Set 2 */


Line1 = "This is line 1";

Line2 = "This is line 2";


/*  Set  3  */

Line1 = "This is line 1";
                          Line2 = "This is line 2";
```

Figure 4-2 End of Line Transfer Example

The first set of PVL assignment statements shows how the statements were intended to appear in the sending environment. The second set shows what can happen when a transfer occurs from a system using carriage return line-feed as end of line to a system using line-feed as the end of line: the lines are double spaced, since the carriage-return and line-feed are each interpreted as an end of line. In the third example, we see that the line-feed character in the receiving environment advances down one line, but does not return to left margin. While the three representations look quite different to the human reader, the parser would interpret each correctly since line-feed and carriage return are both white space characters.

There may be problems with a text transfer which changes the line delimiter if the provided set of octets is indicated by an octet count as in some delimitation techniques in SFDU methodology. The problem would arise because the number of octets would change as the end of line marker changes between a one or two character indicator. If this is the case, the count will be wrong, and the outside mechanism will present a set of octets that ends too early, in the case of going from a single character indicator to a double character indicator, or that includes extraneous material when the transfer is in the reverse direction.

4.4.2 End of Line Deletion

A troublesome problem may occur when moving from system where the line information is imbedded in the internal file structure, (e.g., a character count at the beginning of each variable length record,) to a transfer media where specific characters are used to indicate lines. In some instances, a binary transfer between systems results in a loss of line delimitation between systems. Note that this type of end of line is outside the scope of PVL, which deals with ASCII text characters. A conformant PVL parser would not be required to treat this type of end of line as white space. It may, however, be a local convention to treat the embedded end of line as if it were a white space character.

In these instances, the ability to preserve tokens may be aided by using an explicit delimiter to end statements and by beginning each line with a blank. The explicit delimiter ensures that statement end is recognized, the initial blank helps to separate items within a quoted string.

4.4.3 Addition of Spurious New Line

Certain text transfer mechanisms establish a maximum line length. If a line is longer than the transfer maximum, then a new line is begun at that point, without regard for the possible corruption of the data. This could result in the insertion of white space in the middle of a parameter name or value and invalidate the statement. If a transfer mechanism with such properties is likely to be used, then it would be prudent to assure that lines are shorter than the maximum dictated by the transfer mechanism.

4.4.4 Character Representation Conversions

If data is being transferred between systems using different character representations, such as ASCII and EBCDIC, text transfers will convert the data to the native character representation in the receiving machine. Since PVL is defined as being in ASCII text, a PVL Module that has been translated to EBCDIC would no longer be conformant to the PVL standard. If it is possible to specify the character encoding in a text transfer, it should be specified as ASCII.

4.4.5 Blank Trimming

Some text transfer systems trim trailing blanks from lines. If these trailing blanks are within quoted strings and are expected by the calling application, problems will arise. If it is necessary to preserve these blanks, and to use the transfer protocol, a specific character or sequence of characters can be used as a marker. This marker would be inserted after the last blank to be preserved and before the end of line. Thus, any blank trimming stops upon encountering the marker. Any recipient application would need to be aware of this convention in order to be able to effectively process a PVL Module using this convention.

4.4.6 Tab Conversion

The tab character may represent a different number of spaces on different systems. If information is to be presented in tabular fashion for clarity, it would be prudent to convert tabs to spaces to ensure that the alignment is as intended. Obviously, even the use of blanks may not preserve the columns if a proportional font is used for display or printing.

4.5 Limitations

There are several types of data expression not supported in this version of PVL. The following sections outline those areas where . The limitations are discussed along with strategies for obtaining functional equivalents for unavailable services.

4.5.1 Use of Embedded Quotes

The use of quoted strings within PVL is accomplished by using pairs of either quotation marks or apostrophes. This allows for apostrophes to be represented in text strings that are delimited by quotation marks and for quotation marks to appear in text strings which are delimited by apostrophes.

A limitation is that quoted strings may not contain both apostrophes and quotation marks. It may be possible to break a quoted string into multiple quoted strings, aggregated into a block structure, to allow for the flexibility required to use both forms of quote string delimiters in a combined form. This solution would be in the form of an application convention, rather than a feature of the language.

4.5.2 Indexed Values

PVL does not provide a standard method for indexed values or indexed parameter names. Applications may set conventions to approximate these capabilities.

4.5.3 NonDecimal Numerics

Within applications, nondecimal numerics (binary, octal and hexadecimal) are often used for masks or flag settings as bit strings. On the other hand, within PVL, the nondecimal numerics are interpreted as numeric values. If it is necessary for an application to receive data via PVL as masks, flags, or complemented representations, then a quoted string should be used. The quoted string will be available to the application in unconverted form for processing by the application.

Non-decimal numbers can be used, but the application must use care to make sure the flag settings are mapped to the decoded integer value rather than to the internal representation of any given machine, since machines differ in their addressing schemes for bit level access. On some machines, the most significant bit in a byte is in position 0, in another it is in position 7. By using care in the design of the application, these differences can be taken into account.

5 REQUIREMENTS AND THEIR RATIONALES

This section lists the requirements that PVL was designed to satisfy and the rationale on which the requirement is based. These requirements are categorized as being either high level or specific requirements. High level requirements are listed in Section 5.1 and specific requirements are listed in Section 5.2.

5.1 High Level Requirements

H1. The language shall provide a method of assigning values to a named parameter.

Rationale: It is a common need to be able to reference values by name. This allows the recall of these values later.

H2. The language shall provide a flexible mechanism for the formatting of data, i.e., it shall not require the fixed alignment of values within the format of the language.

Rationale: This frees data producers and users from having to follow rigid formats. It also lessens the impact of transfer induced differences in white space.

H3. The language shall be human readable.

Rationale: This will permit the language to be read by human operators using standard word processors and displays.

H4. Correct parsing of the language by a receiving application shall be possible by use of the language alone and shall not require external communication with the generating application.

Rationale: The language shall be sufficiently well specified that it will always be interpretable in the same way, independent of system environment. This means that the data will not be subject to change when transferred between different environments.

5.2 Specific Requirements

S1. The language shall convey information using assignment statements.

Rationale: The use of assignment statements provides a reliable unambiguous syntax for associating values with names.

S2. The language shall be expressed using a defined character set as follows:

S2.1 The alphanumeric character set (**a-z**, **A-Z** and **0-9**), space, plus the following punctuation symbols:

! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [\ ] ^ _ ‘ { | } ~

Rationale: This character set will permit human readability.

S2.2 The characters specified in S2.1 shall be encoded according to the ASCII standard (ISO 8825).

Rationale: Use of ASCII coding permits unambiguous interpretation in different environments.

S2.3 The following ASCII codes will also be permitted in the language, but shall be treated equivalently to the space character (ASCII code 32):

Horizontal Tab	ASCII code 09
Line Feed	ASCII code 10
Vertical Tab	ASCII code 11
Form Feed	ASCII code 12
Carriage Return	ASCII code 13

Rationale: These format effector values are often used to promote human readability when statements are displayed using standard utilities or word processors.

S3 The language shall be able to convey one of the following types of information for each of its parameters:

S3.1 a character string
S3.2 a numeric
S3.3 an unordered list of any combination of the types defined in the language.
S3.4 an ordered set of any combination of types defined by the language

Rationale: Space scientists, engineers and data processing specialists need to convey the above types of data.

S4 The language shall allow the grouping of statements into named aggregations.

Rationale: There is a need to be able to express complex data structures and/or to associate a group of statements.

S5 A numeric shall be defined as being one of the following:

S5.1 Decimal Integer

S5.2 Hexadecimal Integer

S5.3 Octal Integer

S5.4 Binary Integer

S5.5 Real number with a floating decimal point

S5.6 Real exponential number

Rationale: These numeric representations are those which are commonly used for science and engineering purposes.

S6 Any value within a statement may be optionally assigned units.

Rationale: It may be necessary to indicate particular units used when conveying science or engineering data. (e.g., a set of temperature readings in K.)

S7 The end of statement delimiter shall be either an explicit terminator, the beginning of the next statement or the end of the provided octet space.

Rationale: This allows the accommodation of a variety of styles in the formation of keyword/value statements. In particular, it provides an easier transition from keyword/value languages which do not use explicit statement delimiters.

S8 A mechanism to supply comments shall be provided.

Rationale: This allows the inclusion of explanatory information and thus can contribute to human readability.

S9 A mechanism to denote date and time values shall be provided.

Rationale: These are commonly used constructs in the scientific community.

S10 A mechanism to denote the end of processable statements shall be provided.

Rationale: This allows additional data to be processed to be included within the same provided octet sequence. This is analogous to the construction allowed in other languages.

ANNEX A - ACRONYMS AND GLOSSARY

Purpose:

This annex defines key acronyms and the glossary of terms which are used throughout this Recommendation to describe the concepts and elements of the Parameter Value Language

ANNEX A

Acronyms

ASCII	American Standard Code for Information Interchange
ASN.1	Abstract Syntax Notation One
CCSDS	Consultative Committee for Space Data Systems
CD-ROM	Compact Disk - Read Only Memory
DED	Data Entity Dictionary
EOL	End of Line
ISO	International Standards Organization
FITS	Flexible Image Transport System
ODL	Object Description Language
PVL	Parameter Value Language
SFDU	Standard Formatted Data Unit
YACC	Yet Another Compiler-Compiler

ANNEX A

Glossary of Terms

Aggregation Block: A named set sequence of assignment statements and/or other aggregation blocks.

Alphanumeric character set: The set of characters comprised of the digits 0 through 9 and the letters a-z or A-Z.

Block name: The name used to identify an aggregation block.

Comment string: A delimited string of character information which provides explanatory information. This string is treated as white space syntactically.

Comment delimiter symbols: The symbols used to delimit a comment string.

Data Element: The smallest named item or items of data for a given application.

Data Entity: A named collection of data elements.

Data Entity Dictionary (DED): A collection of semantic definitions for data entities.

Format effector: A control character which effects the display of the printable characters. The format effector characters consist of the horizontal tab, vertical tab, carriage return, line feed and form feed.

Numeric: A special case of unquoted string which conforms to formation rules for numeric values.

Parameter Name: The name used to associate a data value with a parameter.

PVL Module: The externally defined octet space in which PVL statements are written.

Reserved Characters: The set of PVL characters which are reserved for special uses. These characters may not occur in parameter names, unquoted strings or block names.

Quote String Delimiters: The symbols (apostrophe or quotation mark) used to delimit quoted strings.

Quoted String: A delimited sequence of PVL characters.

Standard Formatted Data Units (SFDUs): Data units that conform to a specific set of CCSDS recommendations for structure, construction rules, and field specification definition.

Sequence: A collection of values in which the order of the enclosed values is significant.

Set: A collection of values in which the order of the enclosed values is not significant.

Units expression: A string enclosed within angle brackets which may follow a simple value, set or sequence.

Unquoted String: A value consisting of a sequence of unrestricted characters.

Unrestricted Characters: The set of PVL characters which may be used to form parameter names, unquoted strings or block names.

White space: One or more space or format effector characters. Used to promote readability between syntactic elements or within the contents of comment or text strings.

ANNEX B - - PVL PARSER IMPLEMENTATION ADVICE

Purpose:

This annex provides additional information about PVL in areas that impact the implementation of automated parsing software.

ANNEX B - PVL PARSER IMPLEMENTATION ADVICE

This Annex is intended to augment the PVL specification (Reference [1]) in areas that impact the implementation of automated PVL parsing tools. The purpose of PVL parsing tools is to translate PVL statements into locally defined data structures that can be accessed by local application programs. The purpose of this section is to provide guidance on issues which impact the functionality of a PVL parser implementation while leaving all algorithmic and data structure issues to the discretion of the local design and implementation team. Section B.1 provides an overview of the area of automated language translation to provide a common set of terms and concepts. Further subsections deal with specific areas where experience has identified common issues or concerns. This annex will be expanded in the future as more implementation experience is gained.

B.1 Overview of Language Translators

Translators are highly complex programs, and it is unreasonable to consider the translation process as occurring in a single step. It is usual to regard it as divided into a series of phases, as shown in Figure B-1. Each phase might communicate with the next by means of a suitable intermediate format. In practice the distinction between phases often becomes a little blurred.

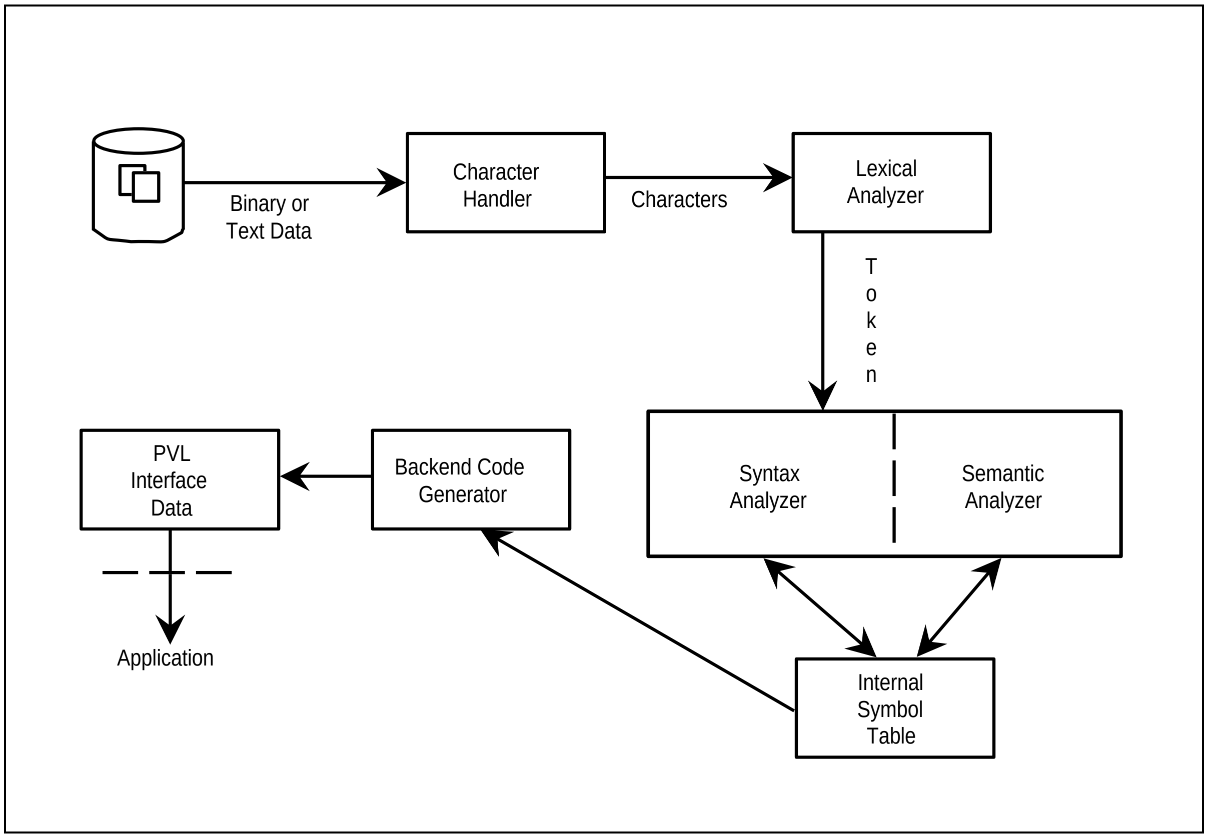

Figure B-1 Language Translator Components

The character handler is the section that communicates with the outside world, through the operating system, to read in the characters that make up the PVL Module. Because character sets and file handling vary from system to system, this phase is often machine dependent.

The lexical analyzer is the section that separates characters of the source into groups that logically belong together to make up the tokens of the language - symbols like identifiers, strings, numeric constants, keywords like **Begin** and **End**, operators like **=**, and so on. These symbols are represented as tokens on the output from the lexical analyzer; although some tokens need attributes with them such as their names or values.

The syntax analyzer compares the sequence of tokens to a defined syntactic structure - it does this by parsing expressions and statements. (This is analogous to a human analyzing a sentence to find components like 'subject', 'object', 'dependent clause' and so on.) Often the syntax analyzer is combined with the semantic analyzer, whose job it is to determine that the syntactic structures make reasonable sense. The division between syntactic and semantic analysis is often blurred. An example of a semantic analysis task in PVL is assuring the values of paired begin-aggregation and end-aggregation statements are equal.

In language translators there is often an extensive code generator to convert parsed language instructions into efficient object code. This function is not necessary for PVL since there is no data manipulation capability in the language. It may however be necessary to translate internal symbol tables to an appropriate format for interface to application programs.

There are a wide variety of compiler building tools available. The most well known are lex (Reference 9) and yacc (Reference [8]). While it is expected that these tools will be used appropriately in local implementation, specific techniques for using compiler building tools are generally considered to be beyond the scope of this Annex, although occasional notes on lex/yacc are included.

B.2 Input Handling Issues

PVL can be transmitted or stored using binary or text format files and protocols. There are two issues that should be noted by the implementor:

- Handling Local End-of-line (text only)
- Handling End-of-PVL-module (text and binary)

B.2.1 Handling Local End-of-Lines

There are a large variety of methods used by computer systems to represent End-of-Line (EOL) in a text environment. While most systems represent EOL by some combination of white space character(s), there are some systems that represent an EOL by a convention such as including a byte count field prior to each line (record) and treating each line as an individual variable length record. If your local system follows such a convention for text files it is recommended that line boundaries be recognized by the PVL Module reader and appropriate PVL white space character(s), such as line feed or the sequence of carriage control and line feed, be inserted into the character string passed to the lexical analyzer. This will ensure that PVL statements, which the user intended to be delimited by line termination characters, will be correctly interpreted. It will also ensure that quoted strings that contain line terminations will not lose semantic value by an unintended concatenation of elements within the quoted string.

The exact implementation of the white space character(s) used as line termination characters is left to the local implementation.

B.2.2 Handling End-of-PVL-Module

The End-of-PVL-Module concept is similar to the End-of-file or End-of-transmission signals in conventional programming languages. The mechanism of terminating the input and exiting the PVL translator are local implementation issues. It is recommended that the PVL Module reader pass a white space character to the lexical analyzer prior to calling appropriate End-of-PVL-module routines to allow for the graceful handling of unexpected termination.

B.3 Numeric Handling Issues

PVL has flexibility in the expression of numeric values. This flexibility creates some issues for the implementors including:

- Resolution of ambiguity between unquoted strings and numerics;

- Optional conversion of numerics to internal machine representations and associated precision issues.

B.3.1 Resolution of Ambiguity Between Unquoted Strings and Numerics

The Abstract Syntax Notation One (ASN.1) specification for all numeric values is a subset of that for unquoted strings. This ambiguity should be resolved by having numerics take precedence over unquoted strings in the lexical analysis phase. The implementation of this is a local issue. NOTE: Lex accomplishes this through the ordering of context rules: if two lex expressions match the same string, then the first of the two expressions in the lex source is chosen.

B.3.2 Conversion of Numerics to Internal Machine Representations

The conversion of numeric values from an ASCII representation to an internal numeric data type is a local implementation decision. In general, the actual range and precision of the ASCII numbers that can be converted will vary based on the host computing system. Therefore, a parser should provide a mechanism for passing an ASCII numeric string that cannot be represented in the internal numeric data types of the host machine. In addition, each local implementation should document:

- The maximum positive and negative integer numbers that can be represented;

- The maximum positive and negative real numbers that can be represented;

- The minimum number of significant digits with which a real number can be guaranteed to have without loss of precision.

The last point is to account for the loss of precision that can occur when representing real numbers in floating point format within a computer. For example, a 32-bit floating point number with 24-bits for the fraction can guarantee at least 6 significant digits will be exact (the seventh and subsequent digits may not be exact because of truncation and round-off errors.)

B.4 Units Expression

PVL allows a units expression i.e., <units> to follow any value (i.e., a simple value, set or sequence) including those that are members of other list values. This introduces some issues in the design of PVL tools, including:

- Allowance for units expression;

- Inheritance or expansion of units expression.

B.4.1 Allowance for Units expressions

A units expression is an optional expression which may follow any value type. Therefore, for each PVL statement, the parser should be prepared to pass to the application the contents of the units expression in addition to the parameter name and value. The contents of the units expression may be any units value (see Section 3.3.2.6). If no units expression is present for a PVL statement, the parser should indicate that fact to the application according to local convention.

In addition, a units expression may follow any value which is member of a set or sequence. Therefore, for each member of a set or sequence, the parser should be prepared to pass to the application the contents of the units expression in addition to the value. Figure B-2 contains an example statement and a conceptual model of a data structure generated by parsing the example. It is meant to illustrate the concept of parsing complex PVL statements, the actual data structures used are left to the local implementor. The example uses a recipe for fruit salad expressed a PVL statement containing nested sets of integers with units to specify the types and varieties of fruit to be cut into one inch cubes for the salad.

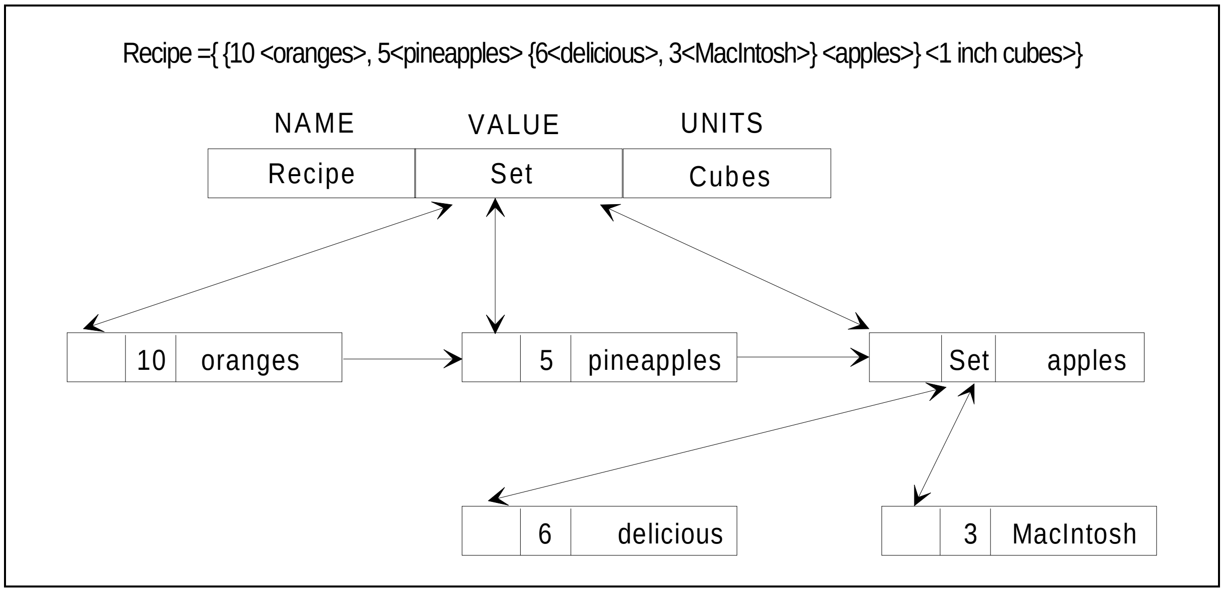
Recipe ={ {10 <oranges>, 5<pineapples> {6<delicious>, 3<MacIntosh>} <apples>} <1 inch cubes>}
NAME
VALUE
UNITS
Recipe
Set
Cubes
10
oranges
5
pineapples
Set
apples
6
delicious
3
MacIntosh

Figure B-2 Conceptual Model of a Data Structure for a List with Units

B.4.2 Inheritance of Units Expressions

The inheritance of units expressions is the domain of the application. An application may wish, for example, that a value inherits a units expression from:

- The previous PVL statement;

- The list in which a value is contained.

This implies that the parser should allow the application access to the units expression following a list prior to accessing the members of that list. The table structure in Figure B-2 follows this constraint by constructing the data structure for the entire statement prior to passing any information to the application program.

INDEX